

JP9-2000-0814

1c821 U.S. PTO  
09/966131  
09/27/01

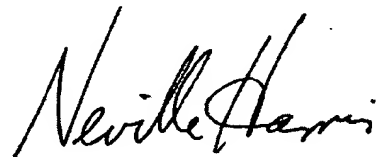
## CERTIFICATE

This certificate is issued in support of an application for Patent registration in a country outside New Zealand pursuant to the Patents Act 1953 and the Regulations thereunder.

I hereby certify that annexed is a true copy of the Provisional Specification as filed on 29 September 2000 with an application for Letters Patent number 507241 made by INTERNATIONAL BUSINESS MACHINES CORPORATION.

Dated 10 September 2001.

**CERTIFIED COPY OF  
PRIORITY DOCUMENT**



Neville Harris  
Commissioner of Patents



**NEW ZEALAND  
PATENTS ACT 1953**

**APPLICATION FOR PATENT**

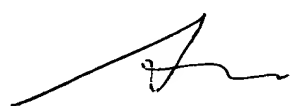
1. We, INTERNATIONAL BUSINESS MACHINES CORPORATION, a corporation of New York, having a place of business at Armonk, New York 10504, United States of America, are in possession of an invention which is described in the accompanying provisional specification under the title "IMPROVEMENTS RELATING TO MULTI-THREAD SYSTEMS".
2. We believe SUNDAR VENKATARAMAN, of London, United Kingdom; PHILIP WONG, of Wellington, New Zealand and SEETHARAMAN VIJAY, of Lower Hutt, New Zealand to be the true and first inventors of the invention, and we are the assignee of the said inventors in respect of the right to make this application.
4. We declare that to the best of our knowledge and belief the statements made above are correct and there is no lawful ground of objection to the grant of a patent to us on this application, and we pray that a patent may be granted to us for the said invention.
6. And we request that all notices, requisitions and communications relating to this application may be sent to:

A J PARK  
Intellectual Property Lawyers and Patent Attorneys  
Huddart Parker Building  
1 Post Office Square  
Wellington  
NEW ZEALAND

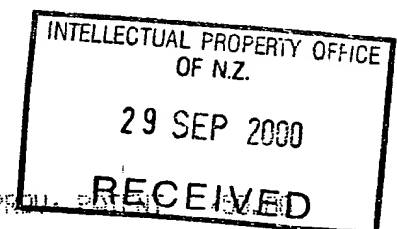
who are hereby appointed to act for us.

**INTERNATIONAL BUSINESS MACHINES  
CORPORATION**

By the authorised agents  
A J Park  
Per:



To the Commissioner of Patents  
LOWER HUTT

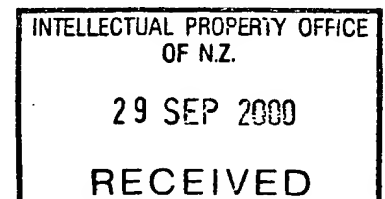


NEW ZEALAND  
PATENTS ACT, 1953

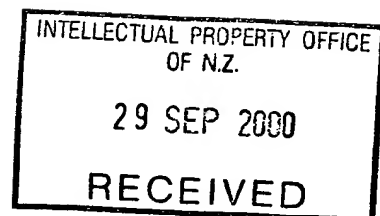
**PROVISIONAL SPECIFICATION**

**IMPROVEMENTS RELATING TO MULTI-THREAD SYSTEMS**

We, INTERNATIONAL BUSINESS MACHINES CORPORATION, a corporation of New York, having a place of business at Armonk, New York 10504, United States of America, do hereby declare this invention to be described in the following statement:



The invention includes any combination of features mentioned herein. Known equivalents of these features are also included.





## Disclosure JP8-2000-1027

Created By: Vijay Seetharaman Created On: 2000/08/18 06:47:30  
Last Modified By: Vijay Seetharaman Last Modified On: 2000/08/30 09:51:07

\*\*\* IBM Confidential \*\*\*

Required fields are marked with the asterisk (\*) and must be filled in to complete the form.

### Summary

Status	Under Evaluation
Processing Location	JP
Functional Area	New Zealand
Attorney/Patent Professional	Hiroshi Ogura/Japan/IBM
IDT Team	Darryl Grennan/New Zealand/IBM@IBMNZ; Richard Burnie/New Zealand/IBM; Francois Esseiva/CanWest/IBM; Darryl Grennan/New Zealand/IBM; Bob Obsieger/CanWest/IBM; Masafumi Takeda/Japan/IBM; Robert Aret/Australia/IBM; Kiyoshi Matsubara/Japan/IBM
Submitted Date	2000/08/29 12:03:39
Owning Division	AP
Select	
PVT Score	To calculate a PVT score, use the 'Calculate PVT' button.
Calculate	
Incentive Program	
Lab	
Technology Code	

### Inventors with Lotus Notes IDs

Inventors: Vijay Seetharaman/New Zealand/IBM

Inventor Name	Inventor Serial	Div/Dept	Manager Serial	Manager Name
> denotes primary contact				
> Seetharaman, Vijay	027673	02/861	015441	Grennan, Darryl D

### Inventors without Lotus Notes IDs

#### IDT Selection

##### Main Idea

##### \*Title of disclosure (in English)

Context based view design to support client side multi-threading

##### \*Idea of disclosure

1. Describe your invention, stating the problem solved (if appropriate), and indicating the advantages of using the invention.

In a typical client GUI application a given view may participate in more than one business case. Creating a view per business case may not be a good solution as it is expensive in terms of memory and resources. In this approach, the view is reused over all the business cases with the help of a view context, which captures all the data elements that need to be presented to the user at any time. A view could be refreshed from a given view context to present new information.

The separation of view contexts from the view also enables the representation of the view information in emerging data representation standards like XML and thus could be easily transferred and persisted.

The view context separates the view from the handlers of a business case and enables different use case handlers to communicate with each other in a multi-threaded scenario.

The view contexts are generally composed of data interfaces of business objects. This helps the views to directly deal with presentation aspects of a business object without introducing any extra layers of data access.

The view contexts group the business objects in the context of presentation in a given view. They also embed the state of the view at the time of the initiation of a business case. It keeps track of the state of the view so that the view can be reconstructed with exactly the same state at a later time.

2. How does the invention solve the problem or achieve an advantage.(a description of "the invention",

including figures inline as appropriate)?

**Details:**

In this invention, views are designed with the basic UI components embedded in them. However, the data used for populating the view is not captured as part of the view itself. This data is encapsulated in the view context. The view presents the data from the view context when it is refreshed with the view context. This way a view can be forced to show the data from the view context. Given a totally new view context, the view would present different information.

**Advantages:**

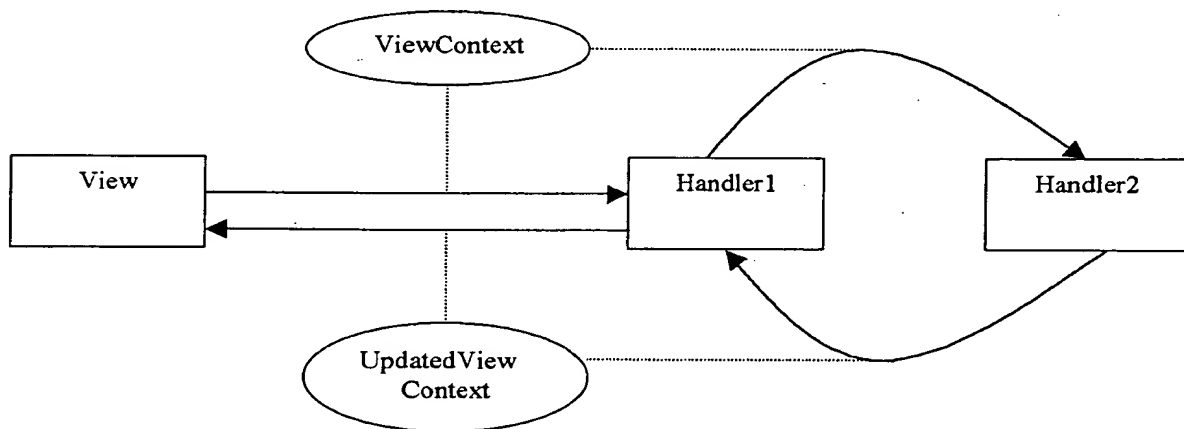
Analogy with the web based design approach:

Also, by representing the view using XML and also the context using XML, the view could be created from data downloaded from the server. The client in this case need not essentially be a browser. It could be a java application. Capturing the state of the view using a context object makes it easier to convert the application later into a web based solution. In such a case, the view would correspond to a XSL stylesheet while the data would be captured as an XML document.

Links the results of a background thread with the handler of the business case:

At the start of a business case, the view passes a newly created view context to the handler of the business case. The view context which is typically made up from a combination of data object interfaces of a given business object implementation would then be updated by the handler during the course of the business case. The handler may choose to forward the view's context to a totally different handler (in order to complete the business case). Finally, the view is forced to refresh from the updated context. The ability to pass the view's context around means there is no state that needs to be maintained by either the view or the handler of a given business case. This is important as creating multiple threads on the client means that control could return earlier than the data. In the absence of a context, the handler has to keep track of the view that it's managing, resulting in a tight coupling between the handler and the view. Thus the context serves to associate the results of execution of a given thread with the handler for the business case.

4. How does the invention solve the problem or achieve an advantage, (a description of "the invention", including figures inline as appropriate) ?



In the above figure, at the start of a business case the view passes its newly created context/existing context to the handler of the business case. The business case handler then can pass it on to a second handler. In a threaded environment, the control would immediately return to the calling handler which then would pass control back to the view. The view then enters the usual event processing loop. Once the called handler completes its task (in a second thread), it posts the reply back (with the updated view context object) to the calling handler. The calling handler can refresh its associated view with the updated view context. In the absence of a context object, the view itself has to be passed around and should be kept track of. Passing the view to a different handler that is totally unrelated to the view breaks the design by making it too complicated and difficult to handle.

3. If the same advantage or problem has been identified by others (inside/outside IBM), how have those others solved it and does your solution differ and why is it better?

Not known.

4. If the invention is implemented in a product or prototype, include technical details, purpose, disclosure details to others and the date of that implementation.

#### View Context:

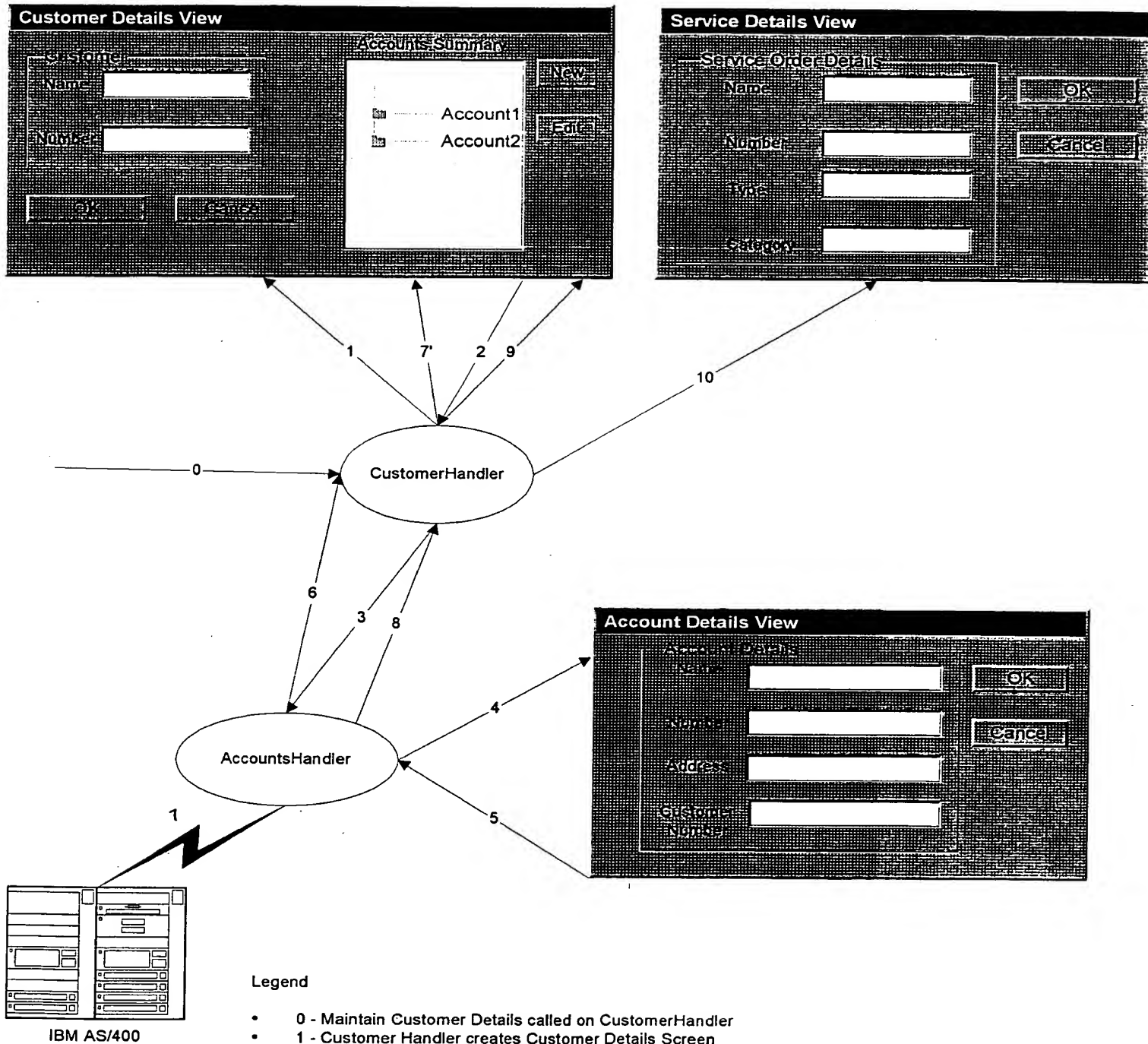
The view context is made up of the data object interfaces for the various business objects. A given view instance understands the make up of a context and can address the individual elements discretely.

In a multi-threaded application, control often passes from one thread to another. Handling controls from one thread to another results in an asynchronous application development paradigm. What this means is that the calling thread posts a request to a worker thread and carries on with its work. The worker thread after completing the request needs to handle control back to the calling thread. This involves the transfer of control and transfer of return data to the calling thread from the worker thread. In such a case, the representation of the view in terms of contexts makes it easier to recreate the view with a new context.

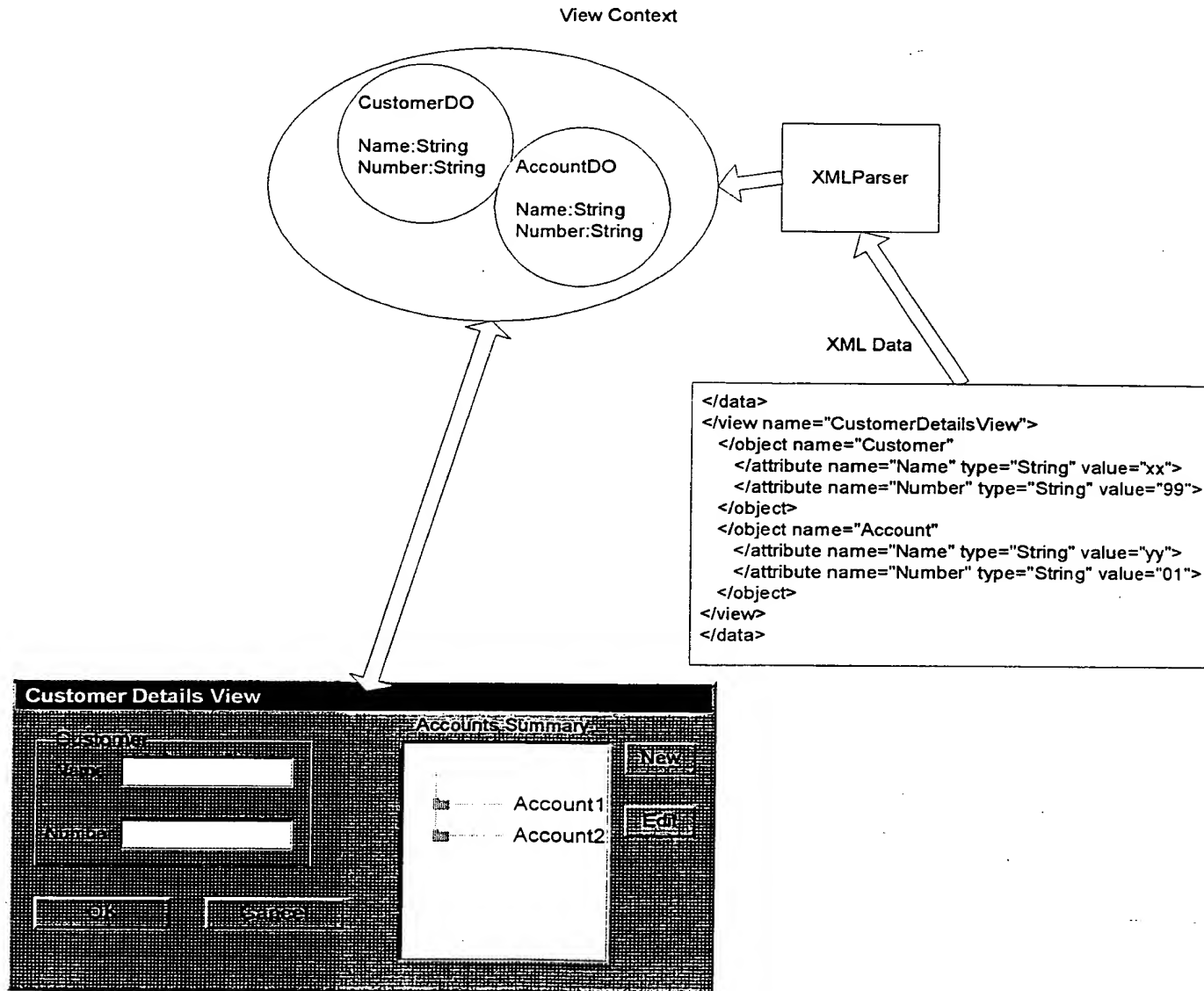
#### Explanation:

In this invention, views are designed with the basic UI components embedded in them. However, the data used for populating the view is not captured as part of the view itself. This data is encapsulated in a view context. The view presents the data from the view context when it is refreshed with the view context. Given a totally new view context, the view would present different information. The view context which is typically made up from a combination of data object interfaces of a given business object implementation would then be updated by the handler during the course of the business case. The handler may choose to forward the view's context to a totally different handler (in order to complete the business case). Finally, the view is forced to refresh from the updated context. The ability to pass the view's context around means there is no state that needs to be maintained by either the view or the handler of a given business case. View contexts exist independently and are not owned by the view or the handler. This is important as creating multiple threads on the client means that control could return earlier than the data. The existence of independent view contexts makes it easy enough to persist the data, represent it in emerging standards like XML.

## Role of View Contexts in Threaded Client Architecture

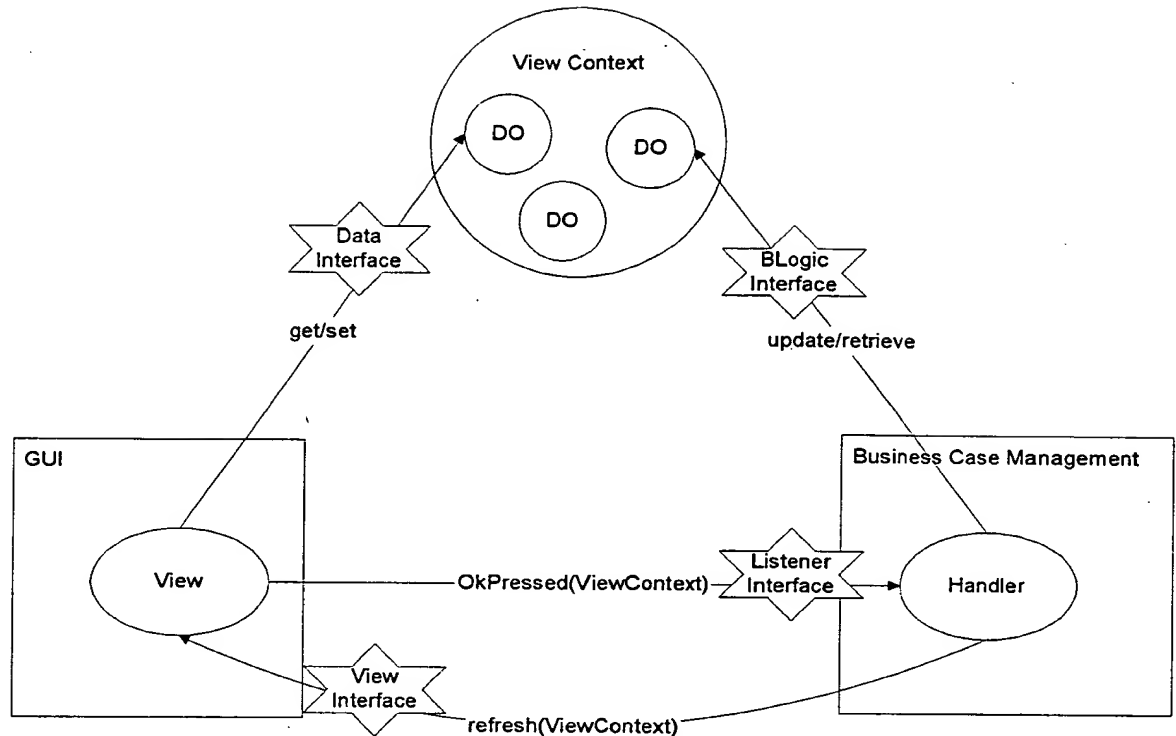


The concept of a separate ViewContext object that captures the complete state of the view enables the data to be transferred as XML information. The client interprets the XML data and gives an object representation to the data in the form of view contexts. The advantage of objects is that they have behaviour associated with them, which could be extremely useful as against message structures, which don't have any associated logic.



The following diagram shows the interaction between the view and the handler using the view context. The view context is a context object, which comes into existence at the time of the creation of a view. The view context

## View-Handler Interaction through View Contexts



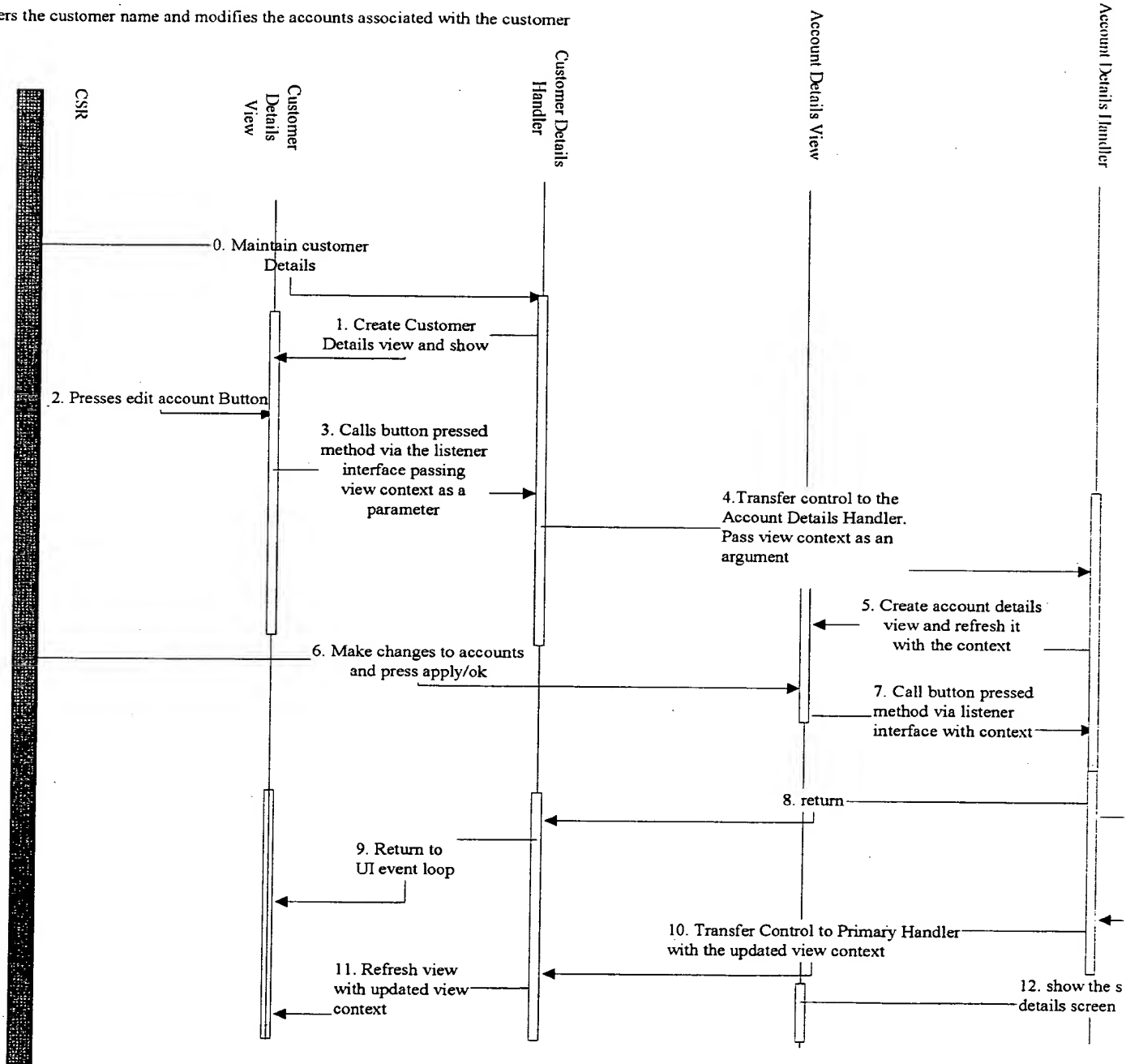
### Legend

1. DO - Data Object

This is explained in the above figure where the customer handler is responsible for showing the CustomerDetailsView, and the ServiceDetailsView. The CustomerDetailsView passes control to an AccountDetailsHandler to handle the sequence of screens that take part in configuring/modifying accounts for a given customer. We see that capturing the state of the view with the help of the view contexts enable the CustomerDetailsHandler to pass the context object around easily and synchronise with the AccountDetailsHandler.

# Object Interaction Diagram

Scenario: The CSR enters the customer name and modifies the accounts associated with the customer



#### View Context:

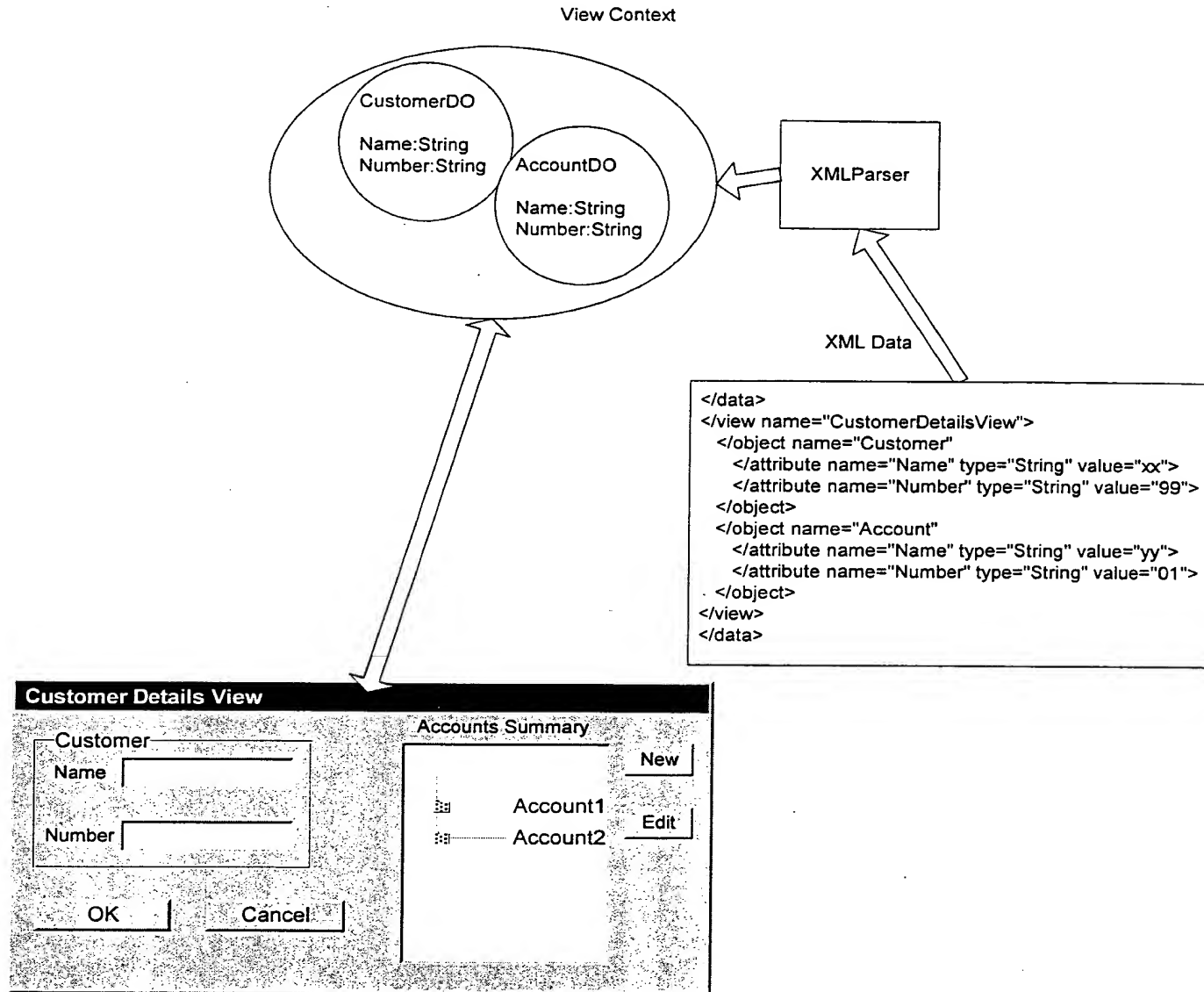
The view context is made up of the data object interfaces for the various business objects. A given view instance understands the make up of a context and can address the individual elements discretely.

In a multi-threaded application, control often passes from one thread to another. Handling controls from one thread to another results in an asynchronous application development paradigm. What this means is that the calling thread posts a request to a worker thread and carries on with its work. The worker thread after completing the request needs to handle control back to the calling thread. This involves the transfer of control and transfer of return data to the calling thread from the worker thread. In such a case, the representation of the view in terms of contexts makes it easier to recreate the view with a new context.

#### Explanation:

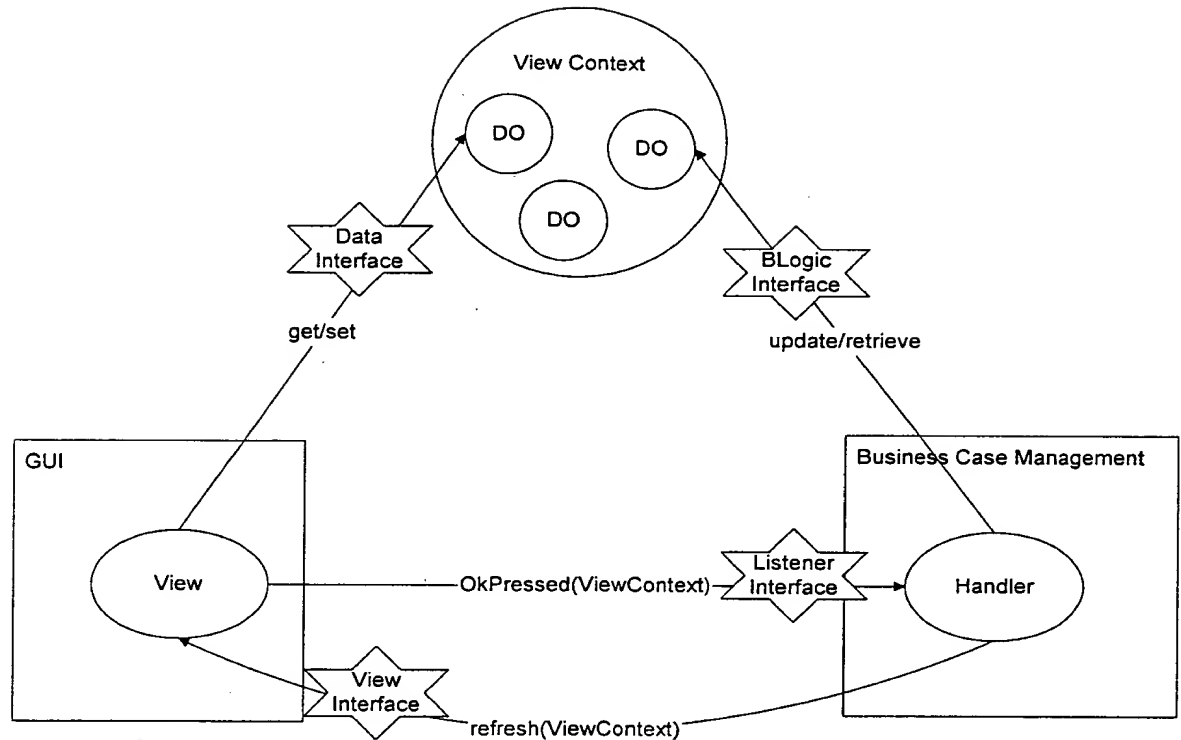
In this invention, views are designed with the basic UI components embedded in them. However, the data used for populating the view is not captured as part of the view itself. This data is encapsulated in a view context. The view presents the data from the view context when it is refreshed with the view context. Given a totally new view context, the view would present different information. The view context which is typically made up from a combination of data object interfaces of a given business object implementation would then be updated by the handler during the course of the business case. The handler may choose to forward the view's context to a totally different handler (in order to complete the business case). Finally, the view is forced to refresh from the updated context. The ability to pass the view's context around means there is no state that needs to be maintained by either the view or the handler of a given business case. View contexts exist independently and are not owned by the view or the handler. This is important as creating multiple threads on the client means that control could return earlier than the data. The existence of independent view contexts makes it easy enough to persist the data, represent it in emerging standards like XML.

The concept of a separate ViewContext object that captures the complete state of the view enables the data to be transferred as XML information. The client interprets the XML data and gives an object representation to the data in the form of view contexts. The advantage of objects is that they have behaviour associated with them, which could be extremely useful as against message structures, which don't have any associated logic.



The following diagram shows the interaction between the view and the handler using the view context. The view context is a context object, which comes into existence at the time of the creation of a view. The view context

## View-Handler Interaction through View Contexts

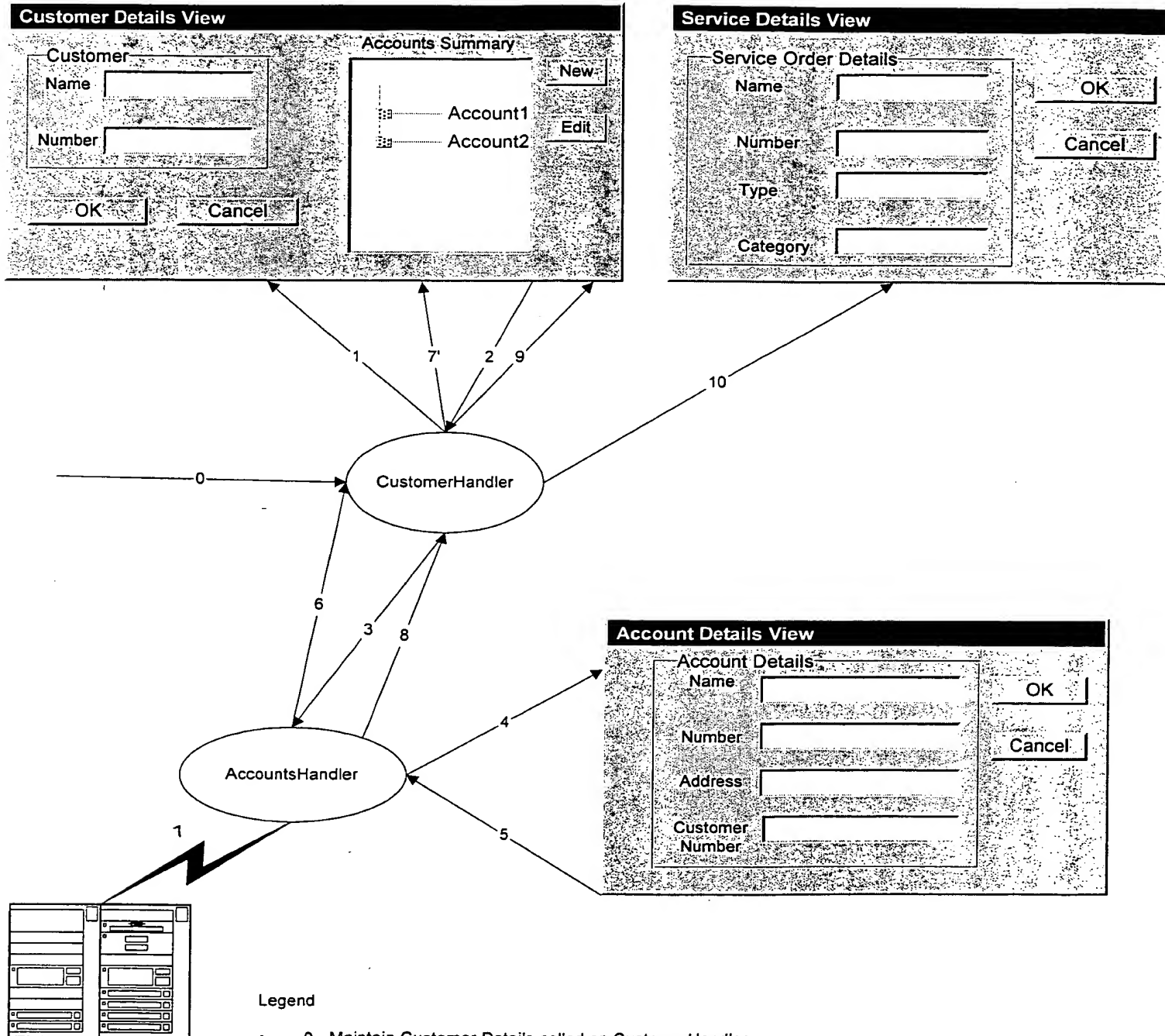


### Legend

1. DO - Data Object

This is explained in the above figure where the customer handler is responsible for showing the CustomerDetailsView, and the ServiceDetailsView. The CustomerDetailsView passes control to an AccountDetailsHandler to handle the sequence of screens that take part in configuring/modifying accounts for a given customer. We see that capturing the state of the view with the help of the view contexts enable the CustomerDetailsHandler to pass the context object around easily and synchronise with the AccountDetailsHandler.

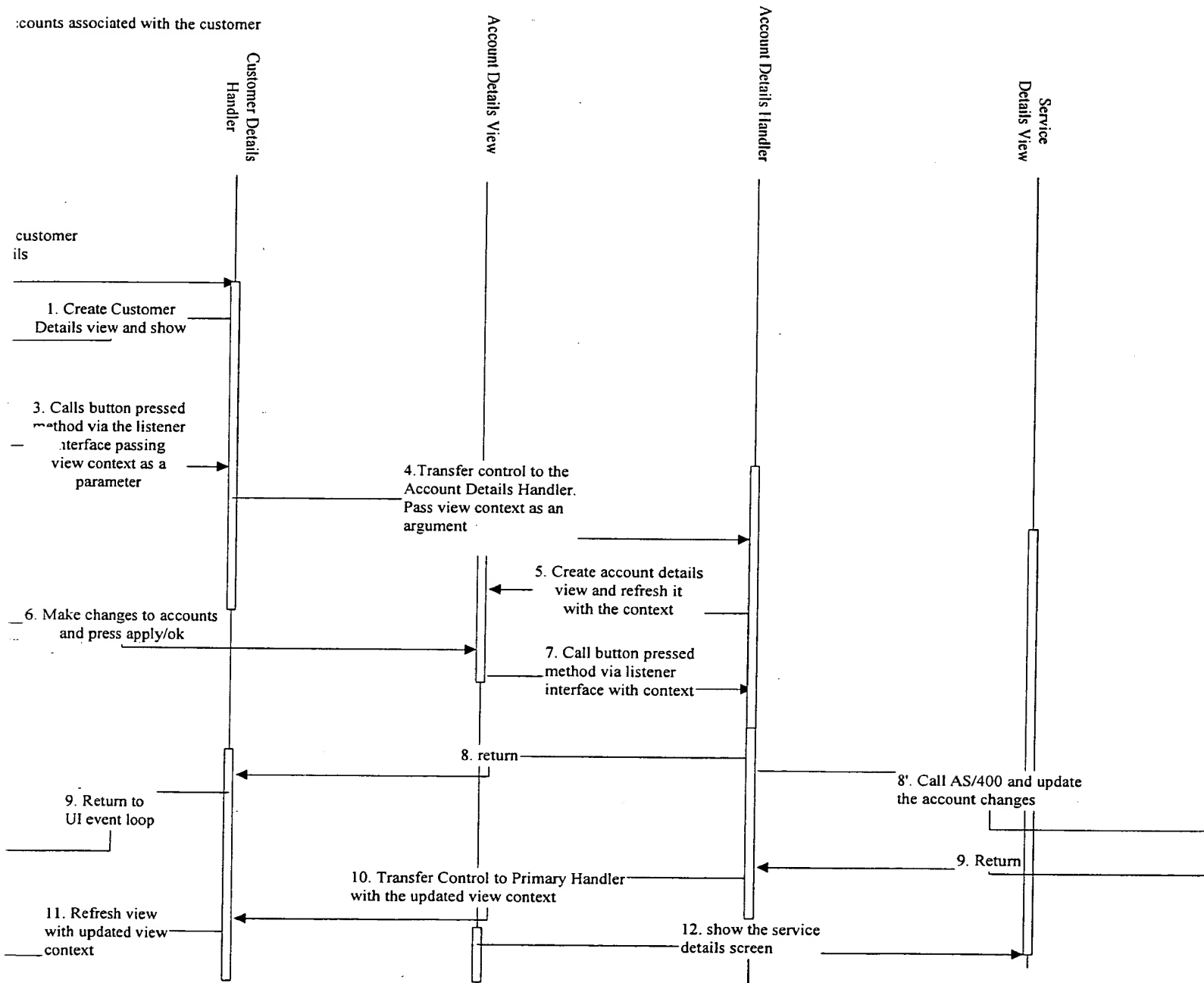
## Role of View Contexts in Threaded Client Architecture



### Legend

- 0 - Maintain Customer Details called on CustomerHandler
- 1 - Customer Handler creates Customer Details Screen
- 2 - User wishes to edit a current account. The view context is passed to Customer Handler
- 3 - Customer Handler passes control to Accounts Handler along with the view context.
- 4 - Accounts Handler creates the account details view and displays account to be edited
- 5 - Control returns to the Accounts handler after the fields have been edited
- 6 - The Accounts Handler uses a language thread and returns control to the Customer Handler
- 7 - The Account Handler persists the changes on the AS/400 server by calling appropriate business logic methods in the business object and by making use of the language thread
- 7' - The Event Thread returns back to the event processing loop from the Customer Handler
- 8 - The accounts handler transfers control to the Customer Handler (in the newly created thread)
- 9 - This control transfer re-synchs the Event Processing thread and the language thread. The language thread is returned to the pool while the event processing thread is used to refresh the Customer Details view of the modified account.
- 10 - Customer Details Handler then creates a new view to create new services (This could be done

accounts associated with the customer





**Software/Release:**

**PCS Number:**

**Problem Type:**

**For:**

**Creation Date:** May 1, 2000

**Owner:** Sundar Venkataraman

**Position Title:**

**Authors:** Vijay Seetharaman, Phil Wong, Sundar V

**Last Amended Date:**

**Amended By:**

**Position Title:**

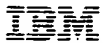
**Category:** Draft – Will change

**Edition:** Draft 0.0, subject to change

**Status:** IBM Internal Use Only

**File Name:** Client Developer Guide.doc

## ICMS Client Java Application Development Guide



© Copyright IBM Corporation 2000 All rights reserved

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of IBM Corporation.

**IBM Confidential**

## Document Control

**Notice**

The official version of this document is the file *Client Developer Guide.doc*. Contact the owner to obtain an official version of this document.

Users of this document are personally responsible for using the official version and for verifying that any copies of the document, in whole or in part, are of the official version. If this document is not a copy of the official version, it is obsolete.

Comparisons with the official document must be made on the basis of the issuance date which is formatted within the official version, and not on the basis of the operating system date associated with softcopies of this document.

**Owner: Vijay Seetharaman****Authors: Vijay Seetharaman, Phillip Wong, Sundar V****Approvers: TBD****Reviewers: TBD**

## Revision History

Date	Revision	Description	Author
May 1, 2000	0.0	First Cut	Sundar V
June 19, 2000	0.1	Document – Version 1	Vijay S

Note: Document version *na* implies that all revisions on the version *n* has been accepted in this version. Generally it is done prior to publishing the version *n*.

## TABLE OF CONTENTS

DOCUMENT CONTROL .....	II
REVISION HISTORY .....	III
<b>1 INTRODUCTION .....</b>	<b>6</b>
1.1 PURPOSE .....	6
1.2 AUDIENCE .....	6
1.3 RELATED DOCUMENTATION .....	6
<b>2 DEVELOPMENT STEPS .....</b>	<b>8</b>
2.1 INTRODUCTION .....	8
2.2 BASE CLASSES .....	8
2.2.1 <i>Background on Interface and Implementation</i> .....	8
2.2.2 <i>Business Object Components</i> .....	9
2.2.2.1 Business Object Implementations (BOImpl) .....	9
2.2.2.2 Data Objects (DO): .....	9
2.2.2.3 Business Objects (BO): .....	9
2.2.2.4 Business Object Homes: .....	10
2.2.2.5 Business Object Remote .....	10
2.2.2.6 Business Object List .....	11
2.2.2.7 Value Object .....	11
2.2.2.8 BusinessObjectRemoteImpl .....	11
2.2.3 <i>UI Related Base Classes</i> .....	12
2.3 PROJECTS AND PACKAGES .....	15
2.3.1 <i>ICMS Customer Care</i> .....	15
2.3.2 <i>Product Management</i> .....	15
2.3.3 <i>ICMS</i> .....	15
2.3.4 <i>IBM Utilities (all IBM packages for utilities)</i> .....	15
2.3.5 <i>IBM ET 400 (AS400 connection)</i> .....	15
2.3.6 <i>Customer Hierarchy</i> .....	15
2.3.7 <i>Usability</i> .....	15
2.4 COMMON REUSABLE IMPLEMENTATIONS .....	16
2.4.1 <i>Scrolling</i> .....	16
2.4.2 <i>Parameters</i> .....	16
2.4.3 <i>Exception Handling</i> .....	17
2.4.4 <i>Help Integration</i> .....	17
2.5 API DESIGN CONSIDERATIONS .....	17
<b>3 CLIENT APPLICATION DEVELOPMENT .....</b>	<b>18</b>
3.1 STEP 1: CODING THE WINDOW .....	19
3.1.1 <i>Step 1A: Layout the Window in VCE</i> .....	19
3.1.2 <i>Step 1B: Externalise Window Literals</i> .....	19
3.1.3 <i>Step 1C: Create Listener Interface</i> .....	19
3.1.4 <i>Step 1D: Add Connections to Listener Events</i> .....	20
3.1.5 <i>Step 1E: Handle local Window Events</i> .....	20
3.1.6 <i>Step 1F: Window Initialization</i> .....	21
3.2 STEP 2: CODING THE CONTEXT OBJECT .....	22
3.3 STEP3 : CODING THE HANDLER .....	22
3.4 STEP 4: EXCEPTION/ERROR HANDLING .....	25

**A TO BE DELETED (ORIGINAL TABLE OF CONTENTS WE WANTED TO HAVE).... エラー!**  
 ブックマークが定義されていません。

<b>B</b>	<b>CLIENT BUSINESS OBJECT DESIGN GUIDELINES.....</b>	<b>26</b>
<b>C</b>	<b>CLIENT DESIGN GUIDELINES:.....</b>	<b>27</b>
<b>D</b>	<b>NEED FOR A CONTROLLER .....</b>	<b>29</b>
<b>E</b>	<b>MVC IMPLEMENTATION STEPS.....</b>	<b>30</b>
<b>F</b>	<b>STEPS INVOLVED IN CLIENT DEVELOPMENT .....</b>	<b>32</b>

# 1 Introduction

## 1.1 Purpose

This document describes the high level design of the client architecture and the steps to build a client application using Java. Readers must be familiar with Java, AS400 API documentation.

## 1.2 Audience

The audience for this document is:

- Client developers with Java coding knowledge

## 1.3 Related Documentation

Refer also to the following documents for additional detail:

- ICMS 5.1 Client Architecture Documentation
- ICMS 5.1 API Architecture Documentation
- ICMS 5.1 Overall Architecture
- ICMS 5.1 Usability Common Non-Functional Requirements document dated 7/10/99
- Java Coding standards
- ICMS Java Style Guide dated 30/3/2000

## 2 Design Overview

### 2.1 *Threaded Client Architecture*

Client applications need to be multi-threaded for the simple reason that UI needs to be responsive even when the user request is being processed. There is a need for doing the data retrieval from the server/database in the background to enable foreground activities like screen refresh, repositioning and re-sizing. Enabling multiple threads could easily complicate the design if too much flexibility is allowed. By limiting what could be done with threads we can easily keep the design simple and manageable.

In this architecture, the following restrictions apply:

- 1) There is only one background thread and different Views should make use of the same background thread to get the data retrieval job done. This simplifies the server design. Multiple threads accessing the AS/400 server in a concurrent fashion would necessitate the creation of multiple server jobs for the same client application. This does not result in a scalable design. Multiple threads using the same connection to the AS/400 would result in commitment control problems on the server as the commitment control is enabled at a job level on the AS/400. Hence partial/incomplete changes done by a thread may get committed as a result of the commit issued by another thread. This results in database inconsistencies. Hence, there is a limitation of one thread doing background jobs. When the thread is processing the data request for a view it is not available for the other views until it is done with the first view. Presently, the implementation does not cater for the queuing of requests from different views. Future implementations would allow queuing of requests from the various views and process them sequentially.
- 2) In the absence of queuing implementation for the background thread, the Views should be designed in such a way that 2 requests are never submitted to the background thread at the same time. This means that the user cannot bring up a different view without dealing with the current view. This means that he cannot have more than one active view at a time. This will be a severe limitation. This can be overcome with the queuing implementation.

### 3 Development Steps

#### 3.1 Introduction

When a client application project commences, a project should be created in VisualAge Java central repository and an owner is assigned. This owner is responsible for allowing additional developers and the access privileges.

Typically a project will have at least two packages, namely the following:

- Com.ibm.icms.new\_application.app This will contain all UI related classes required for the application.
- Com.ibm.icms.new\_application.test. This should contain all the unit test drivers that would be used for testing the business objects or UI. This package is only used internally during development or regression testing.

#### 3.2 Base Classes

##### 3.2.1 Background on Interface and Implementation

Java language allows the implementations of the methods of a class to be protected from the user modifying it. One mechanism available is exposing the methods of a class in the form an *Java Interface*.

For instance, say a Customer object, has some state information (like firstName, title, lastName, customerType) and business logic in the form of methods (like formatName, listAccounts).

Principle :

- Do not allow the state information to be made available to the users of the class directly. Define them as private.
- Define setters and getters (VA Java can generate them) for this state information.

In order to make the object usable for others and control the availability of the methods they can call on this object, use the Java *Interface* mechanism as illustrated in the example code in section ???

### 3.2.2 Business Object Components

#### 3.2.2.1 *Business Object Implementations (BOImpl)*

Business Object Implementations are the ultimate class that stores all the relevant state information and implements business logic or delegates it to the server. A Business Object implementation class is defined as an object with methods that directly implement business logic or map to an underlying API on the server which implements the business logic. Identifying the business objects and relating them to the server data entities, is a subject on its own right and is not dealt with here.

In general, they come into existence either through another business object implementation type or its Business Object Home Type. Business objects implementations have independent database definitions and can be maintained separately. Business Objects implementations can manage their persistence automatically. They have "update", "create" methods that define database definitions. Examples of this are:

- CustomerImpl
- HierarchyImpl
- HierarchyNodeImpl

In general most of the methods implemented in the BOImpl will be defined in a collection of interfaces. Some of the methods will not be defined in any interface and may be protected and other classes in the same Package can access it.

All the methods that are made available in Data Objects and Business Objects must be implemented in this class. In addition, the class may implement other interfaces as appropriate.

#### 3.2.2.2 *Data Objects (DO):*

As discussed before, the state information is made available to the users of the object in the form of setters and getters. Data Object is simply a Java *Interface* that only makes these methods available. Examples are:

- CustomerDO
- HierarchyDO
- HierarchyNodeDO

In some situations, other methods may also be made available in this interface. In a later section it will become clear as to what is the underlying principle used in making these exceptions.

#### 3.2.2.3 *Business Objects (BO):*

All the business methods that are not simply returning the state information from the local variables are defined in this interface.

Some of the example BO's are:

- CustomerBO
- HierarchyBO
- HierarchyNodeBO

BusinessObject is a Java *Interface* over the BOImpl.

### 3.2.2.4 Business Object Homes:

Business Objects are managed by Business Object Homes. Homes are normally used for the following:

- To retrieve a list of BOs based on search criteria. The result is returned as a BOList type (defined in a later section).
- To find a specific instance of BO either from the local cache or from the server database.
- To create a new instance and get reference to this instance. This is typically done when a new customer is created on the client and the values are entered by the CSR. Obtaining a reference does not guarantee the existence of a business object on the server. It provides a convenient way of creating, obtaining specific information about the referenced business object.

Principle:

- All the business objects must be created through a method in a BOHome. Avoid using *new BO()* method to create new instance.
- The primary reasons for this are:
- Recycling objects.
  - Potential for the home to manage instances uniquely if required

Just like the Business Objects having BO and DO as interface to BOImpl, BOHome is also an interface to BOHomeImpl.

Examples are:

- CustomerHomeBO, CustomerHomeImpl
- HierarchyHomeBO, HierarchyHomeImpl

### 3.2.2.5 Business Object Remote

As mentioned before, any method that is not implemented by the BO itself, is implemented in the server. The BO invokes these methods on the server through a peer object called BORemote. Since there are various implementations to invoke a method on the server and the BO itself is not concerned with the implementation of the access to server, this is delegated to BORemote.

In an ideal implementation, the BORemote will be a Java interface and different implementations, BORemoteImpl will implement the access to the servers. This is only a guideline. If the client and the server has tight coupling, the BORemote need not be an interface.

Examples are:

- CustomerRemote and CustomerRemoteImpl
- HierarchyRemote and HierarchyRemoteImpl
- CustomerHomeRemote and CustomerHomeRemoteImpl

### 3.2.2.6 *Business Object List*

BOList is a base class that implements some basic functions for scrolling on the list and getting more items from the list when the caller has iterated over all its retrieved elements. Every BOList has to have a parent. It is the parent that allows addition and deletion from the list. Hence, the BOList is not editable in most of the circumstances. The *addition* and *deletion* of an element from the BOList must be invoked on the parent with the specific instance reference from the list. The list also provides iterating methods.

In general a BusinessObjectHome may have many collections or references to BOLists.

The same principle as explained in the previous section about Java Interface applies here. BOListImpl implements all the methods and BOList and BOEditableList are the interfaces. BOList knows how to retrieve the subsequent objects based on the same search criteria and the filtering values from the server through its parent. To aid in this, it uses two basic state information types namely

- BaseFilter
- ScrollToken

### 3.2.2.6.1 Base Filter

The developer should extend this class for using in each situation. The primary purpose of this class is to allow any filtering information provided by the caller initially, to be used in subsequent calls to the server when next set of business objects are obtained.

### 3.2.2.6.2 Scroll Token

In general, the developer would use this class as is in BOList. This stores the state information about the set that was retrieved. The information will be used in the next set retrieval to position on the server. Since the user of this class should not have to set the values of ivFirstElement and ivLastElement they are treated as byte array field.

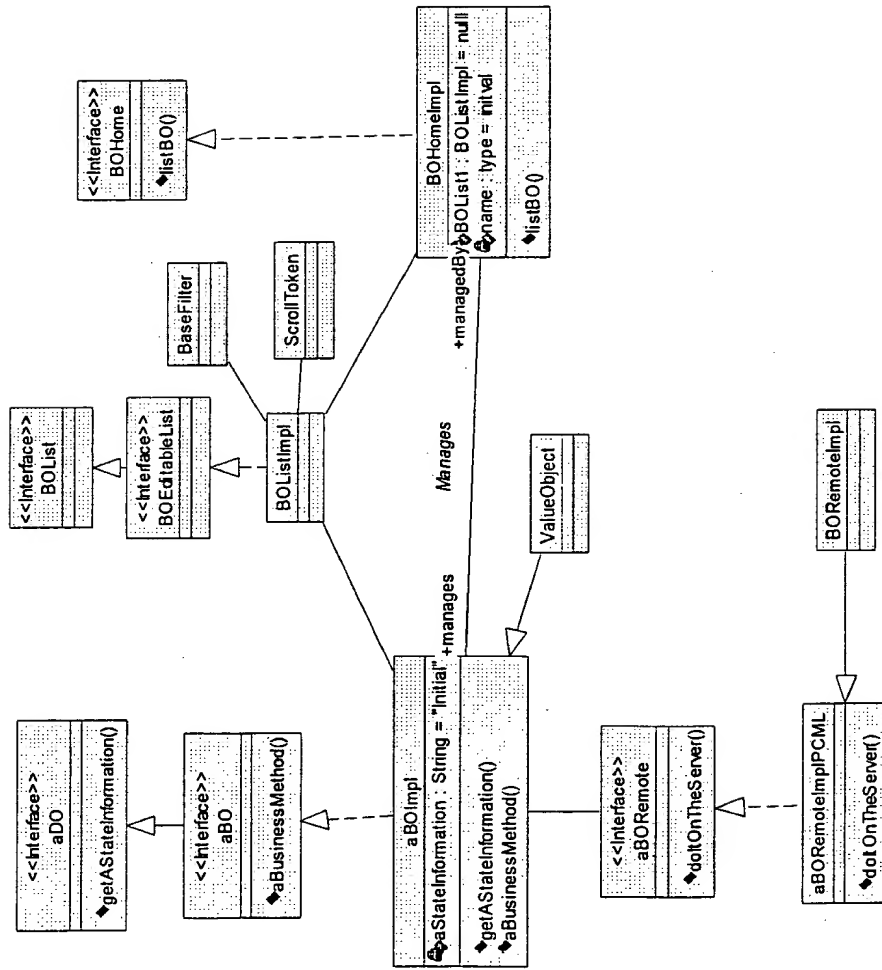
### 3.2.2.7 *Value Object*

All BOImpl must inherit from this base class. This class provides basic operations to keep the object persistence status in relation to the server.

### 3.2.2.8 *BusinessObjectRemoteImpl*

All concrete Business Object Remote implementation classes must inherit from this class. This provides basic error handling capabilities.

The following diagram puts the above definitions in context:



### 3.2.3 UI Related Base Classes

The UI basically consists of two major classes, namely View and Handler. See the appendix for details on the need for these. Handler is a piece of code that controls the flow between views (screens) and deals with the Business Object layer to get information or invoke some business logic. Views are passive. In general they should not have knowledge of what view precedes them or follows them.

Views must handle all the screen-related events that does not require any new business information or creation of new views. For instance, movement of cursor from one field to another is handled by the view. There would always be at least one event that the view itself can not handle and passed to the handler. Typical examples are OK, Cancel, Next, Previous, Help buttons.

Handler and the view pass information using *Context* object. If the view needs to pass handling a particular event to the handler, it

- either converts the particular event or converts into an event that the handler expects
- attaches a context with it
- fires the event

The handler will create view using the following steps:

- Create new instance of the view
- Initialize – typically set the owning handler, and any other non-viewable state information to be set
- Refresh – update the state information and request the view to repaint

Handler receives the event and invokes the necessary method. As discussed before, Views can only access the DO (say CustomerDO) of a Business Object (say CustomerImpl) to get the state information. If it requires refreshing the data and hence, appropriate server logic needs to be called or business logic to be executed, these are passed to the handler.

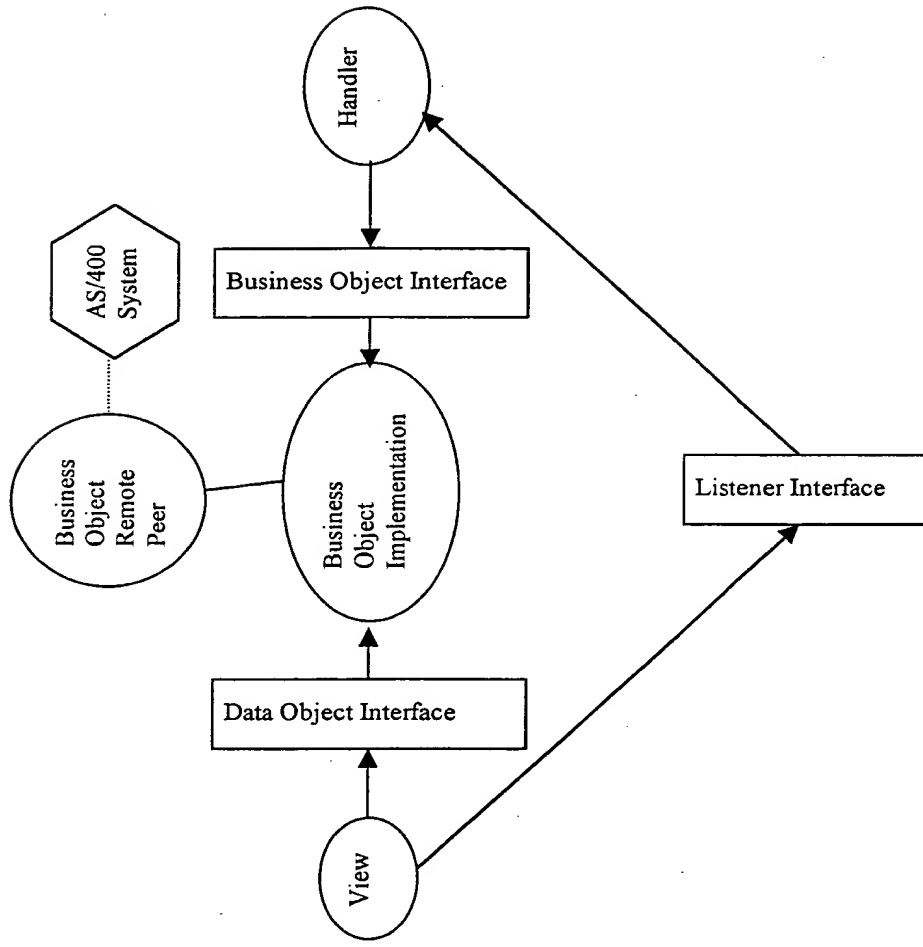
Handler generally has access to methods defined in the BO (say CustomerBO). If a handler needs invoke another view which is generally used by another handler, then the following sequence must be used:

- call the postRequest to the second handler. The handler may do the work in the same thread or another thread.
- Implement a method in this handler to receive the response called postResponse.

Describe the following classes with relevant interfaces using the object model:

- BaseContext
- BaseDialog
- BaseFrame
- BaseHandler
- BusyGlassPane
- CustomTableUI
- MultiLabel
- StatusBar

The following diagram displays the layers in context:



### 3.3 Projects and Packages

All business objects VisualAge Java does not allow a package to be present in two projects on a single workbench. Hence, the business objects belonging to an ICMS component must be kept in a separate project named with component.

In 5.1, since the *Customer Hierarchy* and *MSPPP configurator* were developed in parallel and the development standard was evolving, the packages containing common business objects placed in each project. The situation must be remedied as explained in the subsequent sections.

#### 3.3.1 ICMS Customer Care

This should include:

- Com.ibm.icms.customercare.businessobject
- Com.ibm.icms.customercare.businessobjectimpl
- Com.ibm.icms.customercare.test

#### 3.3.2 Product Management

- Com.ibm.icms.productmanagement.businessobject
- Com.ibm.icms.productmanagement.businessobjectimpl
- Com.ibm.icms.productmanagement.test

#### 3.3.3 ICMS

- Com.ibm.icms.parameters.businessobject
- Com.ibm.icms.parameters.businessobjectimpl
- Com.ibm.icms.businessobject
- Com.ibm.icms.businessobjectimpl
- Com.ibm.icms.ui
- Com.ibm.icms.util
- Com.ibm.icms.test

#### 3.3.4 IBM Utilities (all IBM packages for utilities)

#### 3.3.5 IBM ET 400 (AS400 connection)

#### 3.3.6 Customer Hierarchy

- Com.ibm.icms.customercare.hierarchy.app
- Com.ibm.icms.customercare.test

#### 3.3.7 Usability

- Com.ibm.icms.productmanagement.msppp.app
- Com.ibm.icms.productmanagement.msppp.test

### 3.4 Common Reusable Implementations

#### 3.4.1 Scrolling

The following example illustrates the coding required to scroll on a customer business object:

- Define CustomerListImpl as

```
public class CustomerListImpl extends BusinessObjectListImpl {
    private CustomerHomeImpl ivParentBOImpl;
}
```

- Define CustomerListFilter as

```
public class CustomerListFilter extends BaseFilter {
    private String ivCurrency = null;
    private int ivCycle = 0;
    private int ivFrequency = 0;
}
```

- Implement a method in CustomerHomeRemote to get the list first time. In this example the signature appears as follows:

```
public BusinessObjectEditableList findByName(String firstName,
    String lastName,
    int cycle,
    int frequency,
    String currency,
    CustomerListImpl aCustomerList) throws BusinessException;
```

Implement a method to retrieve the next set in CustomerHomeRemote.

- Implement the method *retrieveNextSet()* in CustomerListImpl. The implementation should call the method to retrieve next set in CustomerHomeRemote.

#### 3.4.2 Parameters

The classes that are used for this are:

- ParameterDO (Java interface)
- ParameterHomeBO (Java interface)
- ParameterHomeImpl

- `ParameterImpl`
- `ParameterHomeRemote`
- `ParameterListImpl`

`ParameterHomeImpl` maintains a hash table of collections of various parameter types. There is a method to retrieve the list of parameters of a certain type as *ParameterListImpl* through `BOList` interface. If the parameter is already retrieved from the server, then the list is simply returned to the user. If this is the first time, a particular parameter type is requested, then it uses the `ParameterHomeRemote` to get the list from the server.

Each element in the `ParameterListImpl` is a `ParameterImpl` object which can be accessed through `ParameterDO` interface.

If there are some parameters that are not standard and has extra attributes, then specific class must be defined inheriting from `ParameterImpl`. See `ParameterNotificationImpl` for an example.

### 3.4.3 Exception Handling

The major classes used are:

- `BusinessErrorImpl` and `BusinessError` (Interface). This encapsulates each error message between client and server that are generated by the application. The error message, code and severity are available in this.
- `BusinessException` manages list of business errors. This is subclassed from standard exception. In general, when a business rule is violated this exception is thrown. In general this will be raised in side `BORemote` classes as this is where the server programs are called. It is possible to raise them in `BO` classes as well, if a business rule is implemented locally.
- The following classes handle the exceptions:
  - `BusinessObjectRemoteImpl`. This handles the exceptions to collect information and then throws it again
  - `BaseHandler`. This is the ultimate class that responsible to display the error to the user when required. It decides based on the severity if the error is to be popped up or shown in the status bar or simply logged.

### 3.4.4 Help Integration

The major classes used is:

- `BaseHandler`. It listens for any help event generated by pressing PF1. This invokes a browser with appropriate URL for the help.

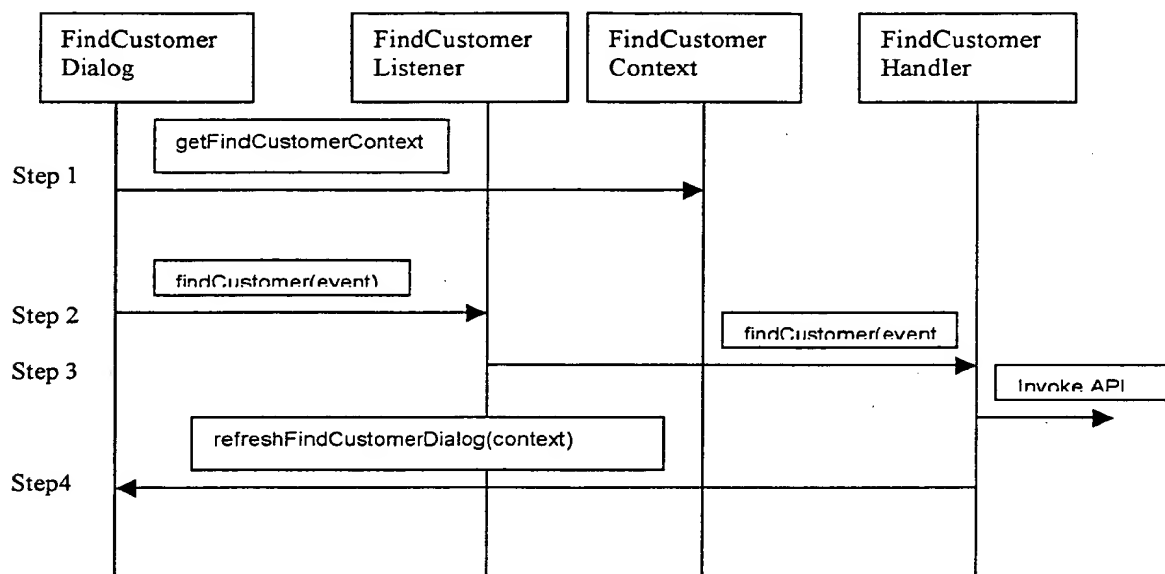
## 3.5 API Design Considerations

- Each API should be committable by itself.
- Do not allow nested scrolling.
- Do not get the same state information of a business object from multiple APIs.

## 4 Client Application Development

Development of the (UI) User Interface portion of the application is best described by an example. The example chosen has been demonstrated in Customer Hierarchy, the scenario involves retrieving a set of customers matching a context. To do this, the User invokes the Find Customer function from the Hierarchy Builder and enters a set of search criteria, a set of matching customers is displayed. The example focuses on development of the find customer function and its invocation from the Hierarchy builder.

The diagram below shows a high level flow of events to retrieve a list of customers for a search criteria entered by the user.



### Step1:

The user enters search criteria and presses the find Button. The FindCustomerDialog creates a FindCustomerContext object and populates the context object with the search criteria .

### Step2:

The FindCustomerDialog fires the findCustomer event which encapsulates the context object.

### Step3:

The FindCustomerListener detects the findCustomer event and forwards it to the FindCustomerHandler for processing. The FindCustomerHandler invokes the appropriate API that retrieves the list of customers.

### Step4:

The FindCustomerHandler refreshes the Window with a list of matching Customers.

The following sections describes in steps how a developer builds the view the associated control classes to implement the Find Customer Function:

## 4.1 Step 1: Coding the Window

**Prerequisite:** The developer must be familiar with the UI Interface style guide for User interfaces. This is found in doc XXXXXXXXXXXXXXXX.

The Window can be a Frame or a Dialog. In the Customer Hierarchy application the primary window (HierarchyBuilderView) is a resizable frame, all secondary windows are Dialogs. The Find Customer window is a dialog.

The developer should use the Visual Composition Editor (VCE) feature of VisualAge for Java to construct windows. Advanced developers may code windows by hand with using the VCE. The advantage of the VCE is that you can visually see what you are developing.

### 4.1.1 Step 1A: Layout the Window in VCE

Use the VCE to layout the Window. Make sure that you are familiar with the UI Style Guide to ensure that fonts, pushbutton sizes and the correct widget types are used.

### 4.1.2 Step 1B: Externalise Window Literals

String literals will be used as labels for the Window title, pushbutton text, static text, etc. These literals should be externalized into a ListResourceBundle class to allow translation to other languages.

Example: ListResourceBundle used in Customer Hierarchy

```
public class CustomerHierarchyResourceBundle extends java.util.ListResourceBundle {
    static final Object[] contents = {
        "Title", "Customer Hierarchy Builder",
        "HelpMenu", "Help",
        "HierarchyMenu", "Hierarchy",
        "HierarchyNewMenuItem", "New",
        "HierarchyOpenMenuItem", "Open",
        "HierarchyPrintMenuItem", "Print"
    };
}
```

**To access items from the ListResourceBundle:**

Example retrieves the text from the ListResourceBundle that matches the "cancelBtn" key

```
cancelBtn.setText(resCustomerHierarchyResourceBundle.getString("cancelBtn"))
```

### 4.1.3 Step 1C: Create Listener Interface

Create a new Listener interface in the View. This will create the necessary classes ( Event Class, Listener Class, and EventMulticaster Class) that will allow the Handler to process business events. The New Listener interface is create in the VCE by selecting the Features menu->New Listener Interface. Add the events that you wish the Listener to handle. In the Find Customer Scenario events that we wish the Listener to detect are:

1. FindCustomer

2. GetNextMatches
3. ShowDetails.

The VCE creates the necessary events prefixed by "fire". These methods can be customized. An example of the fireFindCustomer method generated.

```

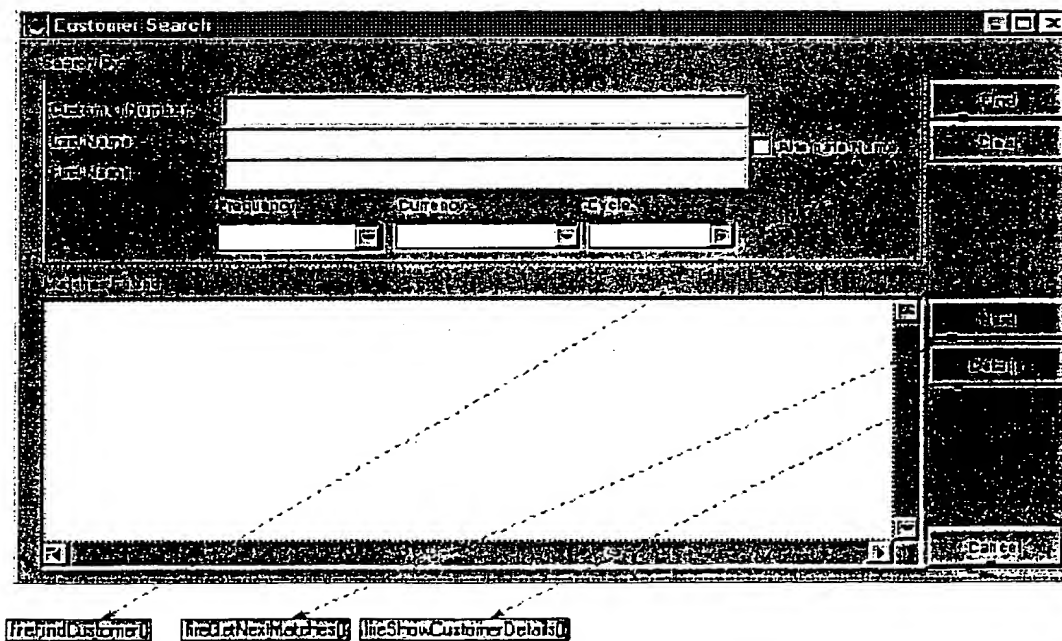
Method to support listener events
@param event com.ibm.icms.customercare.hierarchy.app.FindCustomerEvent
protected void fireFindCustomer(com.ibm.icms.customercare.hierarchy.app.FindCustomerEvent event)
{
    if (findCustomerListener == null) {
        return;
    }
    findCustomerListener.findCustomer(event);
}

```

#### 4.1.4 Step 1D: Add Connections to Listener Events

Add connections to widgets that require a server API to be invoked:

The connections are indicated by the dotted lines.



#### 4.1.5 Step 1E: Handle local Window Events

Simple events that do not require invocation of Server APIs should be handled locally within the view. An example is the User pressing the Clear pushbutton to clear data entered and retrieved from a previous search.

These events are handled in the *actionPerformed* method in the Window, For Example:

```
public void actionPerformed(java.awt.event.ActionEvent e) {
    if (e.getSource() == getClearBtn()) {
        clear();
    }
}
```

#### 4.1.6 Step 1F: Window Initialization

Perform any window initialization in the *initialize* method. This may include setting default button states, setting default values in lists and combo boxes.

The initialize method is regenerated each time window is modified in the VCE. To avoid losing user code ensure this code is placed begin:

// user code begin

.....  
some code

// user code end

Sample *initialize* method in FindCustomerDialog:

```
private void initialize() {
    try {
        setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
        setTitle(resCustomerHierarchyResourceBundle.getString("CustomerSearch"));
        setSize(656, 390);
        setResizable(false);
        setContentPane(getBaseDialogContentPane());
        initConnections();
        setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
        addExceptionHandler();
    }
    //user code begin (2)

    getRootPane().setDefaultButton(getFindBtn());
    getFindBtn().requestFocus();

    //set default values in combo boxes
    getCurrencyCbo().setSelectedItem(null);
    getCurrencyCbo().repaint();
    getCycleCbo().setSelectedItem(null);
    getCycleCbo().repaint();
    getFrequencyOfCallingCbo().setSelectedItem(null);

    //user code end
}
```

**Rule:** All Frames should be subclassed from BaseFrame.

**Rule:** All Dialogs should be subclassed from BaseDialog.

**Guideline:** Use a Layout Manager to ensure that the window can be sized correctly.

**Guideline:** Minimize the use of connections within the visual builder. Too many connections clutter the builder screen and the code generated is inefficient.

**Guideline:** Typically use connections to connect actions that require events to be handled by a Handler.

Validation:

**Guideline:** Syntactical validation should be performed in the view. Examples of syntactical validation include:

1. Field Length
2. Field Type ie. input field that accepts only numeric characters.

**Guideline:** Semantic validation should take place on the server. This encompasses all business rule checking.

**Guideline:** GUI actions that do not trigger server API's should be handled by the window itself. These actions are typically triggered by implementing an action listener for the particular event.

## 4.2 Step 2: Coding the Context Object

Context objects are used to pass information between the view and the Handler. The Context object is usually encapsulated within the Event object that gets triggered from the View.

In the Find Customer scenario, a FindCustomerContext object is used to hold the search criteria for the search and the result list for the matching customers.

The following code segment shows a definition of the FindCustomerContext

```
public class FindHierarchyContext extends BaseContext {
    public static final int OPEN = 1;
    public static final int EDIT = 2;
    public static final int END = 3;
    public static final int NONE = 4;

    public static final int CUSTOMER_SEARCH = 1;
    public static final int HIERARCHY_SEARCH = 2;
    public static final int ALL_SEARCH = 3;
    public static final int ACTIVE_SEARCH = 4;
    public static final int INACTIVE_SEARCH = 5;

    private int ivAction = NONE;
    private int ivSearchType = ALL_SEARCH;
    private String ivCustomerNumber;
    private String ivHierarchyName;
    private int ivSelectedIndex = 1;

    private BusinessObject ivResultList = null;
    private HierarchyDO ivSourceHierarchy = null;
}
```

**Guideline:** A Context object should be used to pass information between view and Handler

**Rule:** All Context objects should subclass from BaseContext.

## 4.3 Step3 : Coding the Handler

The Handler is responsible for processing satisfying business requests that are generated by the View. In the Find Customer Example, the business requests that require Handling are:

1. Find Customer
2. Retrieve Next Customers
3. Open Customer Details.

**Rule:** All Handlers subclass from the BaseHandler.

**Rule:** The Handler is responsible for invoking server API's.

**Rule:** The Handler creates the view and associates any require listeners to the view.

```
private void createFindCustomerDialog() {
    if (FindDialog == null) {
        FindCustomerDialog aFindCustomerDialog;
        aFindCustomerDialog = new FindCustomerDialog(HierarchyBuilderView.getParentView());
        aFindCustomerDialog.addFindCustomerDialog();
        aFindCustomerDialog.showCentered();
        aFindCustomerDialog.addFindCustomerListener(this);
        aFindCustomerDialog.addErrorListener(this);
        aFindCustomerDialog.addHelpListener(this);
    }
    try {
        aFindCustomerDialog.show();
    } catch (Exception e) {
        System.out.println("Exception occurred in main() of Java Swing JFrame.");
        e.printStackTrace(System.out);
    }
}
```

**Rule:** Handlers can create other Handlers.

In our scenario, the Handler for the main window (HierarchyBuilderHandler) creates the FindCustomerHandler

```

/**
 * Invokes the Find Customer API, the event object
 * transports the FindCustomerContext which contains the search criteria and
 * results of search
 *
 * @param event <code> FindCustomerEvent</code>
 * @see FindCustomerEvent, FindCustomerContext
 */
public void findCustomer(FindCustomerEvent event) {
    setFindCustomerEvent(event);
    setBusy(true, true);
    Runnable request = new Runnable() {
        public void run() {
            BusinessObjectEditableList boList = null;
            FindCustomerContext criteria = getFindCustomerEvent().getFindCustomerContext();
            String cycle = criteria.getCycle();
            String frequency = criteria.getFrequencyOfBilling();
            try {
                CustomerHomeBO homeBO = CustomerHomeBO;
                InitialContext ctx = com.ibm.commerce.customer.pare.businessobject.mpl.CustomerHomeImpl;
                boList = criteria.getResultList();
                criteria.setResult(true);
            } catch (BusinessException e) {
                criteria.setResult(false);
                handleBusinessException(e);
            } catch (Throwable th) {
                criteria.setResult(false);
                handleException(th);
            }
        }
    };
    Runnable response = new Runnable() {
        public void run() {
            refreshFindCustomerDialog(getFindCustomerEvent().getFindCustomerContext());
            setBusy(false, true); // chain to parent
        }
    };
    com.ibm.commerce.util.ThreadSupport.getInstance().invoke(request, response);
}

```

The standard pattern used to invoke a Server API is demonstrated in the code segment above. Two Runnable objects are created one to hold the request and the other the response. The invoke method queues the request and response objects to be run sequentially on the ICMS ThreadSupport object. Presently, only one request will be satisfied at a time. The ThreadSupport object creates a separate worker thread to first run the request runnable object. Once the request runnable object returns, it queues the response runnable object on the Java event dispatching thread. This is necessary as the updates to UI should always be done by the event dispatching thread as swing components are not thread safe. The event dispatching thread then refreshes the UI with the results. In future, more than one request could be queued with the ThreadSupport object. We may assign threads from a pool to do the job or strictly serialise the execution of all request runnable objects. This serialisation is needed because of AS/400 toolbox mechanism of creating one AS/400 job per connection to the AS/400. If a single AS/400 job is accessed by several threads then problems relating to commitment control may occur. Hence, it's always safe to limit the number of worker threads to only one and queue all background processing requests (just like the event dispatcher thread).

Runnable objects are created to improve responsiveness of the user interface by running each Runnable on a separate thread.

Refreshing of the User Interface should take place in the response object. In our scenario the FindCustomerDialog is updated via the refreshFindCustomerDialog passing the FindCustomerContext which contains the matching customer list

**Guideline:** `BusinessObjectEditableList`, and `BusinessObjectList` should be used to represent lists of objects that and to be displayed in the View

#### ***4.4 Step 4: Exception/Error Handling.***

Error handling occurs in the `BaseHandler`, all business errors are displayed to the user in a `ErrorDialog` while warning messages are displayed in the application's status line.

The VCE creates a `handleException()` method. This method needs to be deleted as exceptions are handled in the base handler.

## A Client Business Object Design Guidelines

### Lazy State Evaluation for the Business Objects:

The business object has a corresponding client side representation of its state. This is achieved with the help of the value objects which form a part of the business object's server side state. As a given business object is made up of a number of data objects it is possible to lazily evaluate the state of the business object by invoking the corresponding retrieve methods on the client. Example: For the Package Business Object, in order to evaluate the summary information, the "retrieveSummary" method is called. To evaluate the Prerequisite list, the "retrievePrerequisiteList" method is called.

### Business Methods in the Business Objects:

Business Objects have methods that correspond to an AS/400 API call via the AS/400 ToolBox for Java. A business method call on the BO is translated into an AS/400 API call via the toolbox. This involves conversion of Java data types into RPG data types (done with the help of the Tool box translator classes).

Summary of business classes and business methods:

#### **PackageHome**

##### **Void Retrieve(PackageKey key)**

Retrieves all packages starting from the key position

##### **Void RetrieveAll()**

Retrieves all the packages

##### **Package nextPackage()**

Returns the next package from the currently retrieved list. The positions are maintained with the help of a context token/cursor inside of the home.

##### **Package previousPackage()**

Returns the previous package from the currently retrieved list. The positions are maintained with the help of a context token/cursor inside of the home

##### **Boolean hasPreviousPackage()**

Returns true if there is a previous package from the current position of the cursor

##### **Boolean hasNextPackage()**

Returns true if there is a next package from the current cursor position

##### **Package intialPackageReference(PackageKey key)**

Returns a reference with key. This method creates a reference or a proxy object locally with the key information. The successful creation of the reference object does not guarantee the existence of the business object in the database. This business object reference could then be used to populate information about the BO by calling appropriate retrieve methods or create new objects by calling the create method.

#### **Package**

##### **Void RetrieveSummary()**

Retrieves the summary information for the business object "Package"

##### **Void RetrieveGeneralInformation()**

Retrieves the general information about a package BO.

## B Business Case Handlers

### Non-Functional Requirement

1. Design should identify elements of reuse
2. Design should take care of separating functionality, where necessary, so that it could be extended easily by adding code and not modifying existing ones.

#### Examples:

The following examples illustrate the elements of reuse.

Consider a "Find" View. This view typically displays some entry fields to capture the search criterion and displays a result set. Now this view could be reused in a number of places. The context in which the view is displayed should be independent of the view. This means that the knowledge that an advanced search window would appear when a customer search is initiated and a simple search window would appear in case of a hierarchy search should not be embedded in the "Find" View itself. If it is embedded in the view then it is reusable only to the extent that all such possibilities of the next window that appears from the "Find" view is known. The design is not extensible if the knowledge is captured in the view.

Separating functionality is an important concept in OO designs. We should be able to identify small reusable objects. This way we increase reusability. We know that the smaller the object is, the more reusable it becomes. However, the potential pitfall here is when do we stop? If we keep making the objects more and more granular they become more reusable but less usable. Practically, it may be impossible to use them at all. Hence there should be a compromise as to what the size of an object is and what its reusability is. We decided to draw the reusable line at the view level than at the view components level. We made sure that each view is made up of smaller and reusable components provided by standard Java Swing Components and also some third party components (e.g., DatePicker).

An application could be thought of as being made up of a number of business cases. A business case may be one of 1) Finding a list of customers satisfying a given search criterion, 2) Creating a new hierarchy 3) editing an existing hierarchy etc. These are typically called as use cases in OO design.

Write down the various steps involved in a business case.

For example:

- 1) In order to create a new hierarchy, the user is presented with a dialog window, which captures the details needed to create a hierarchy.
- 2) When the user presses OK a call is made to the server to create the new hierarchy and the UI on the parent window is refreshed to show the new hierarchy.

These steps are then translated into the screen flow diagram and the object sequencing diagrams. The screen flow diagrams shows what screen follows what in the business case. The object sequence diagram shows what objects interact to get the job done. In order to separate the functionality, the business case is captured as a handler. Once the application identifies the business case, it would create an appropriate handler and pass control to it. In fact, the application itself could be thought of as a handler.

The handler governs the UI flow, i.e., creates views in sequence, creates appropriate business objects and invokes business methods in them, handles the errors appropriately and disposes off the window.

The window is responsible for presenting details of a given business object (its state information) and populating the state of a given business object (captures the entry details for a given business object). However, it does not understand the operations performed by users from a business perspective. For example, pressing the OK button, would indicate to the handler that the details entered for a given business object should be committed. However, the view itself doesn't understand the meaning of the OK button pressed. However, the view may understand drag and drop events and other events that relate only to the

view. If the drag and drop results in a business event then it is passed on to the handler as it may trigger a totally new business case. This way we classify events as business events and view events. View events are handled by the view. The handler governs the business events. This is important, as a mix of the two would result in an inefficient design. The advantages of separating the two are as follows:

- 1) Same business events triggered by different views could be targeted to the same handler. For example, consider two entirely different windows having a help button or a find button. If we know that pressing the help button/the find button would bring up the same business case then the business event could be re-targeted to the same business case handler. This is important as we are trying to follow a use-case based approach and one use-case can extend a given use case or be part of a larger use-case. Modeling a use-case, with handlers, results in very good reuse of the handlers.

## C Client Design guidelines:

The view captures the data portion of a business object(s). The business object exports the data object interface and the business object interface. The data object interface is used by the view for presentation purposes. The data object interface is also called the presentation interface of the business object. The business object interface captures the business logic. The central idea is to capture all the data object interfaces (relating to a given view) into a single object called the View's context object (also known as the View Data Model object). The primary reason for the introduction of the context objects relates to the threaded client design. Because of the presence of a GUI thread and a corresponding background thread, the GUI events are totally detached from the background work. In order to relate a given GUI event to a background job and to trace back, a state or some context is necessary. Any view, in this design, could be reinstated if the context is known. The context is like a snapshot of a view data. The view can recreate itself given the context. The view can register the last user operation in the context. The handler registers the result of the last operation in the context. The view takes appropriate actions based on the results of the last operation.

In response to a user event, the View constructs a business event by associating its context with the business event. The view may store specific information (like the operation performed) in the context that may not be visible to the handler. The view then fires the event at its listeners. Since the handler is a listener, it is notified of the business event that happened in the view. The handler then handles the business event by calling appropriate business methods on the business object. The handler executes the business methods using the background thread. Once the execution completes, the handler can register the results of the operation with the View Context and can refresh the view with this updated context. Thus with the help of context objects the view and the handler interact and also synchronize.

On the other hand, if the business case is an extension of another business case the handler may decide to post the request to the handler of the other business case. It has to first convert the event that it received into an event that the other handler recognizes. On completing its task, the second handler, posts a reply back to the first handler (which is registered as the parent handler of the second handler) passing back the updated event (should be called the handler context). The original handler can then refresh the view with the new updated context. In this design, the calling handler is not aware of the views that the called handler might bring up. This is very essential in a good design as they clearly separate handlers too.

## D UI Implementation Steps

From visual builder do the following:

- 1) Define accessors for fields that may be accessed from outside the view.
  - Choose methods tab from the visual composition editor
  - Press the methods button to create a new method
  - Type in the method name, choose method parameters and modifiers
  - Define the method body.

Example:

```
Public final void setCode(String str) {
    GettfCode().setText(str);
}
```

```
public final String getCode() {
    return gettfCode().getText();
}
```

- Repeat the above steps until all relevant accessors are defined.
- 2) Define Listener interface for relevant GUI events that cannot be handled by the view internally.
    - Choose Features / Create new listener interfaces and within that create methods to handle each event. One listener interface for group of events
      - PackageDetails - interface with methods PakcageDetailsOKed, PackageDetailsCancelled, PackageDetailsApplied
  - 3) Use the visualBuilder to locate the components that will generate the events. For instance choose, OK, Cancel, Apply. Then in the Event-to-code interface of the component, simply select the corresponding event-fire method to invoke for the corresponding component events (ie. A button, will generate action events. The event-fire method should be linked to the action-event).
  - 4) Define handlers for the view. The handler should also implement the interfaces provided by the view inorder to be able to listen to the events generated in the view. The handler can implement all the interfaces or there could be separate handlers for each of the interface.
    - Define a new class using the class button in the workbench view.
    - Define this class to implement the interfaces provided by the view. This will generate the skeleton code for all the interface methods.
    - Define an instance variable of type same as the view intialised to null.
    - Define a getter method which would create an instance of the view if not created already and shows the view. Example:

```
private PackageDetails ivjPackageDetails1 = null;
private PackageDetails getPackageDetails1() {
    if (ivjPackageDetails1 == null) {
        try {
            ivjPackageDetails1 = new com.ibm.icms.usability.msppp.PackageDetails(); //
            create an instance of the view and show it.
            ivjPackageDetails1.setVisible(true);
            // user code end
        } catch (java.lang.Throwable ivjExc) {
            // user code begin {2}
            // user code end
        }
    }
}
```

```
        handleException(ivjExc);
    }
};
return ivjPackageDetails1;
}
```

- Define an initialise method which would add the handler as a subscriber to the interfaces exported by the view. Example:  
Void initialize() {  
 GetPackageDetails1().addPackageDetailsListener(this);  
 GetPackageDetails1().addScrollListener(this);  
}
- Write the bodies for the methods in the listener interface. This method, eg., PackageOKed, is supposed to call server APTs and populate the views using the accessor methods provided by the view.

## E Steps Involved in Client Development

Item No	Description	Method1 V (view – API)	Method2 MV	Method3 MVC
1	Creation of panels done using VisualBuilder.	X	X	X. No extra lines of code.
2	Create interfaces			X For each externally handled event (dealing with an API) 1 line per event. Events could be logically grouped into interfaces.
3	Create getters/setters for the view data model to be available for the handler			X For each getter/setter we may need 1-5 lines
4	Handlers externally defined			X. No extra lines of code.
5	Create Business Objects		X	X. No extra lines of code.
6	Calling the API	X	X	X. No extra lines Of code
7	Number of objects created		X. model objects created	X. One extra handler object per view. Model objects created.
	Productivity			
	Maintainability			
	Quality			
	VisualAge for Java way of doing it			
	Alignment with AVI stuff			

International Business  
Machines Corporation

By the authorised agents

A. J. PARK

Per

DRAFT, subject to change

IBM Internal Use Only

